CHI: A General Agent Communication Framework[1]

Laurence R. Phillips, Steven Y. Goldsmith, & Shannon V. Spires

Sandia National Laboratories
Albuquerque, NM 87185
lrphill, sygolds, svspire@sandia.gov
505-845-8846, -8926, -4287

Abstract: We have completed and exercised a communication framework called CHI (CLOS to HTML Interface) by which agents can communicate with humans. CHI follows HTTP (HyperText Transfer Protocol) and produces HTML (HyperTest Markup Language) for use by WWW (World-Wide Web) browsers. CHI enables the rapid and dynamic construction of interface mechanisms. The essence of CHI is automatic registration of dynamically generated interface elements to named objects in the agent's internal environment. The agent can access information in these objects at will. State is preserved, so an agent can pursue branching interaction sequences, activate failure recovery behaviors, and otherwise act opportunistically to maintain a conversation. The CHI mechanism remains transparent in multi-agent, multi-user environments because of automatically ganerated unique identifiers built into the CHI mechanism. In this paper we discuss design, language, implementation, and extension issues, and, by way of illustration, examine the use of the general CHI/HCHI mechanism in a specific international electronic commerce system. We conclude that the CHI mechanism is an effective, efficient, and extensible means of the agent/human communication.

Introduction

Communication underlies and forms a context for community. Although the World-Wide Web (WWW or just "the Web") forms an extensive digital community, communication mechanisms are often laboriously hand-built, application-specific and not very flexible. Agents promise to alleviate this difficulty by providing personal guides that either understand humans, understand their environments, or both; at a minimum, such agents will understand how to communicate using the available media: e-mail, fax, and most importantly the Web.

The key to making this work in practice is the establishment of appropriate abstraction barriers and the factoring of capability into more-readily maintained components. In particular, we advise against agents with monotlithically embedded comunication mechanisms. A plug-in or mix-in capacity to use whatever media may be available is more robust and easier to manage and maintain. This approach encapsulates the means of communication and keeps it independent of the agent's reasoning means. The means—methods specialized on multiple inputs, classes modifiable at runtime, multiple inheritance—are available in modern dynamic languages. This approach also scales well across the important dimension of language type. We examine this issue during discussion of the international electronic commerce system, in particular how information from several sources in several private business dialects is incorporated into the mechanism.

We have developed CHI (CLOS-to-HTML Interface), a class/method hierarchy that enables communication between agents and humans. The information moving between the user's web browser and the agent during communication consists of objects transformed into Hypertext Markup Language (HTML). These objects are "carriers" of information and allow the user to have the illusion of direct interaction with the agent. The agent, on the other hand, relies on a set of independent performatives, such as those expressible in KQML ([Finin et al. 93], [Finin and Labrou

[1] This work was performed at Sandia National Laboratories, which is supported by the U.S. Department of Energy under contract DE-AC04-94AL85000

97], and [Finin et al. 97]). The conversion between an internal object representation and KQML would allow agents to communicate with one another in much the same way that CHI enables communication between agents and humans. This paper focuses on agent/human communication; however, we discuss extension to other communication channels in the “Extension to other languages” section.

How CHI works

From the user's point of view, the user hits the “submit” button of a form or clicks an anchor, the browser says “Host contacted; waiting for reply ...”, and shortly a (new) page is displayed on the user's browser.

The agent with whom the user is communicating must receive the message sent by the user (which implies that some connectivity must exist between users and agents in multi-user multi-agent situations), generate a response specifically for that user (which implies that the message must be identified as having come from the user), and send the response back to the user (which implies that the agent must have some connection with the user).

CHI must therefore enable the agent to transmit a page to the user's browser in response to each form submission or anchor click (POSTs and GETs, in HTTP terms). In this paper, we use the term “post/get” to refer to any such transmission.

This is accomplished by the following cycle, illustrated in Figure ##:

1. CHI creates and initializes a page object instance.

   The appropriate class is instantiated based on the class of the message object being transmitted. Message component objects are similarly converted into sub-page elements.

2. CHI translates the page object instance into an HTML stream.

3. CHI entrains the ACGI mechanism to the intended recipient’s input channel.

3. The recipient takes some action that returns a message to the CHI application.

   The Web server receives the GET/POST and sends it to the CHI application via the ACGI mechanism.

4. CHI extracts the identifier from the message and uses it to acquire the object instance from which the page instance was generated.

5. CHI places information from the message into the page instance.

6. CHI executes relevant methods and functions to produce the next page instance. At this time, objects can be created, changed, placed into, or retrieved from the session; and information from these objects can be used to formulate the next page and/or to control the application.

7. CHI links the old page instance and the new page instance together to maintain session continuity.

8. The cycle begins again with the new page instance.

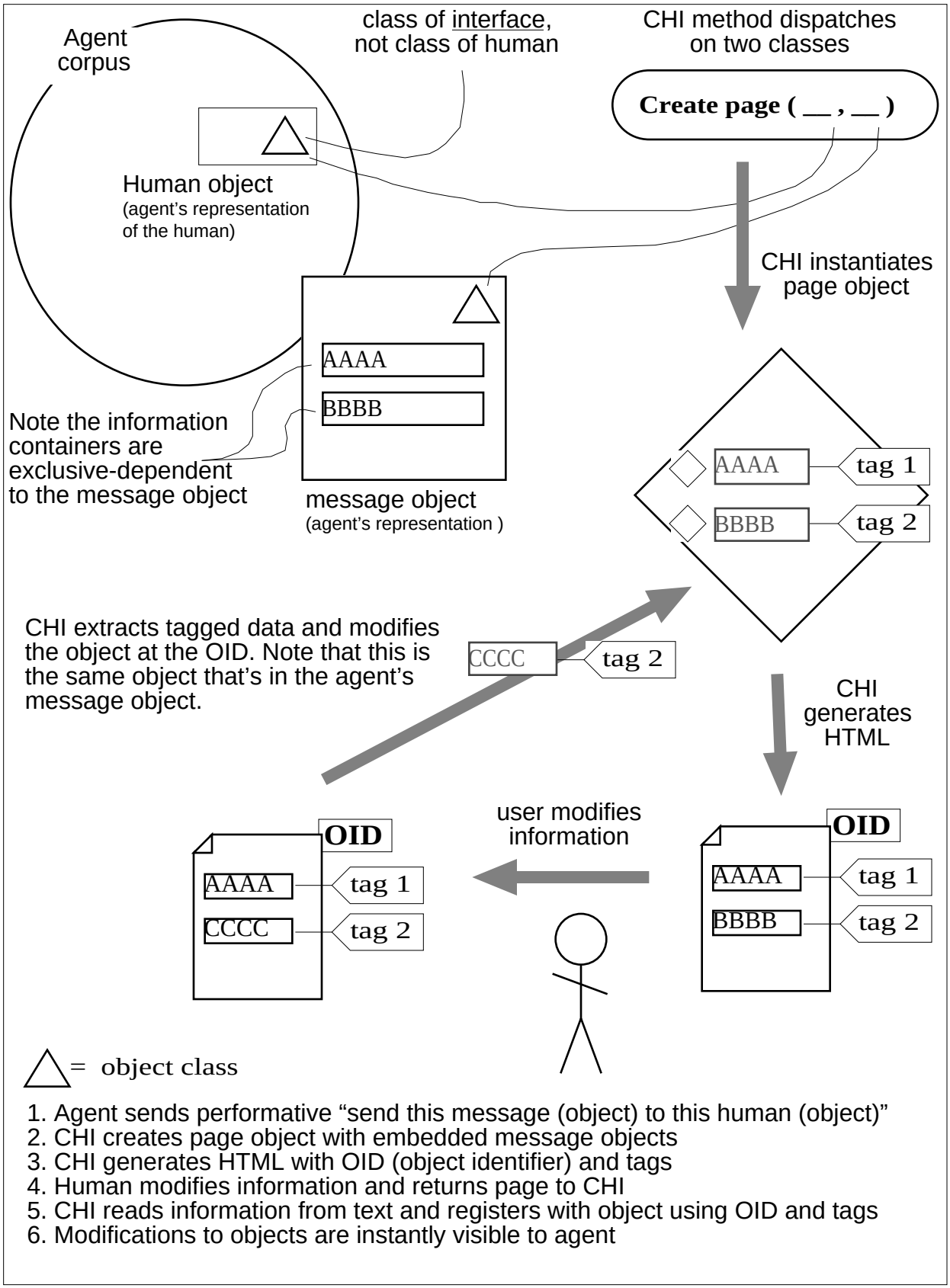


Figure 1. The CHI information cycle from Agent to human and back

Developing applications using the CHI methodology

CHI is a set of classes and methods. Instantiation of the CHI classes creates objects that are able to render themselves as HTML. The highest-level objects result in pages or frames whose components contain the information being communicated. An agent can read or update this information at will. Nested objects are called recursively and generate serial HTML that maintains the nesting. The HTML contains tags and names that enable CHI to identify and update the source objects when information from a CHI-generated page is returned to the server. Input sites in the HTML are automatically named so that CHI can retrieve the individual information values and update the original objects.

CHI automatically maintains program state and provides continuity for multiple simultaneous user sessions by linking together dynamic objects that represent pages and sessions. This solves the problem of operating a stateful application with a stateless browser. CHI supports workgroup computing on the Web because the CHI process maintains multiple user states simultaneously and non-synchronously.

The designer of an agent communication interface has two primary tasks: (a) To define the class structure of the objects that constitute the interface and (b) To write the runtime methods that:

- Translate information from the user into a syntax that the agent understands,
- Send a performative (or performatives) to the agent that contain the translated information,
- Receive the agent's response,
- Convert the agent's response into HTML, and
- Return the HTML to the user.

The design methodology is to create a set of classes, initialization mechanisms, and methods that "glue" the agent and the user's Web browser together. CHI provides base classes and functionality. The designer creates classes that inherit from and specialize the CHI classes to add look and feel, application-specific organization, run-time results, and so on.

CHI provides the builder of a communication interface a set of classes whose instances can convert themselves into HTML. To assist in this process we have developed a software mechanism that converts an HTML structure into a class definition and an initialization function that creates the appropriate nested CHI instances. This mechanism is called HCHI (HTML to CHI). In this paper we focus on the CHI mechanism and reserve disscussion of HCHI.

The designer of a CHI application specializes these classes, writes methods for initializing their instances, and defines methods for handling the possible inputs. These define the structure and behavior of the browser pages that the user will see. The actual page contents are determined, and the page instances constructed, at run time. CHI thus turns the Web into a dynamic computing environment.

Interfaces with many interactive and inter-related Web pages can be created in a few hours. sufficient information to enable an object-oriented framework internal to the agent to instantly and accurately register with and reference its internal representations of the human and the relevant instance variables on the page, and the variables' current values. This process is automatic; the agent merely receives message notification and updated objects without further reference to HTML or HTTP syntax, which is fully encapsulated within CHI. Formulation of a response is also entirely internal to the agent and independent of HTML. At output time the agent merely emits a response (from its point of view) and CHI generates a new web page instance dynamically, so there is never an html file anywhere in the system (with the possible exception of an initial template, which can also be generated by an agent at the initiation of a contact).

Extension to other languages

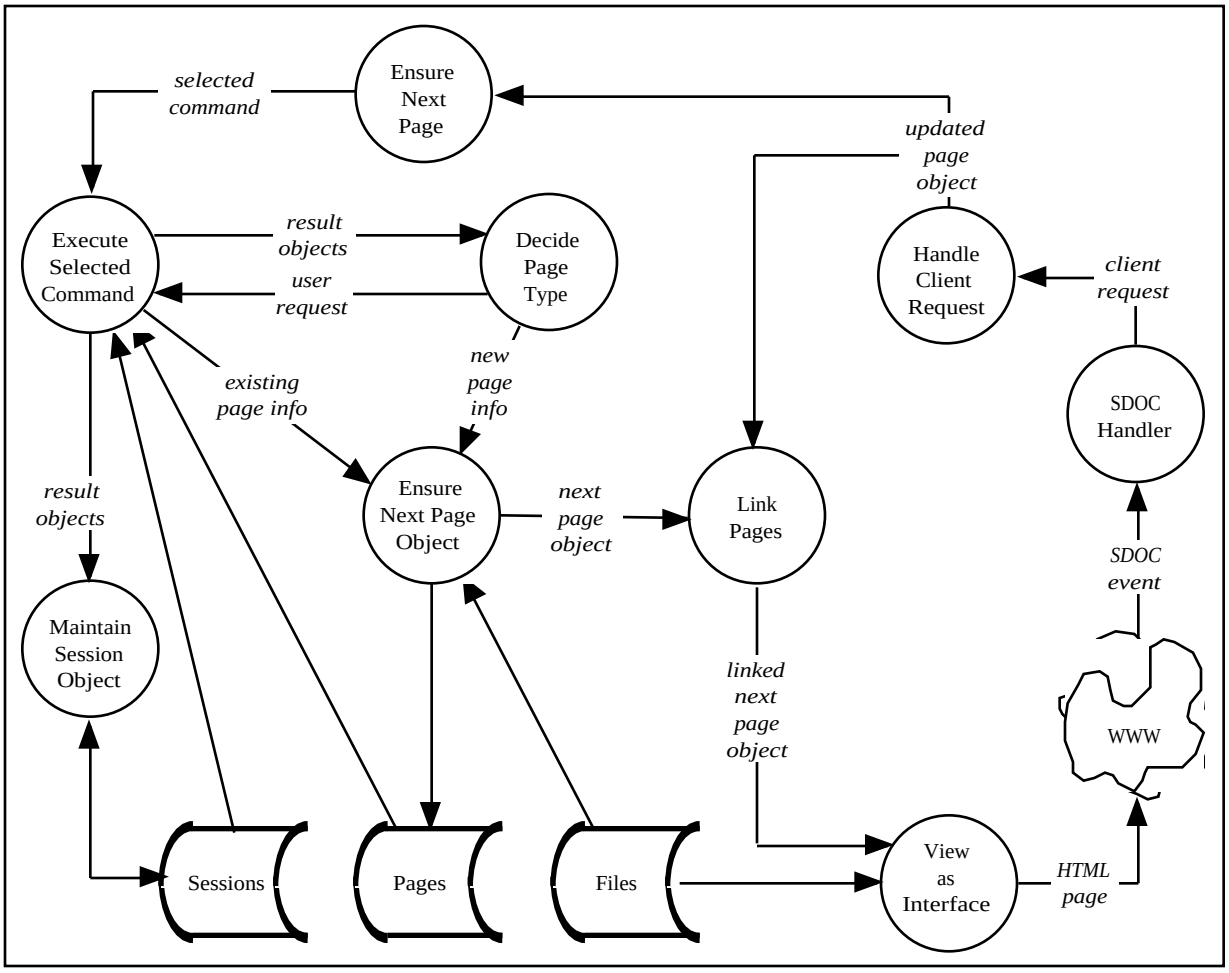
CHI provides a set of foundation classes that represent pages, forms, and widgets. Instances of these classes translate themselves into HTML at run-time for Web consumption. CHI classes are hierarchically organized and available to the designer for specialization, along with their inherited methods.

Updating CHI itself to support changes in HTML or HTTP, or extending CHI to wholly different interface languages, is conceptually straightforward because the language specifics are factored into methods specialized on the interface class. This also means that different versions of HTML and different interface languages can be supported simultaneously by a single running CHI server.

At this time, agents equipped with CHI can communicate only with humans. We have used multi-methods, however, which allows extension to classes of recipients other than humans.

The link between the agent's own internal language and that of the recipient resides in multi-methods, that is, methods specialized on multiple classes. The specializing classes are the output message class and the recipient's interface type (see Figure 1). Invariant semantic content is embedded into a stream of the appropriate syntax as the message object is converted by the appropriately specialized method regardless of the language.

We expect to create a mechanism similar to the existing HTML method but based on KQML to enable agent-to-agent communication. This task would consist in creating a message for each message class of interest but specialized on the new interface class. The task should proceed independently of the agents' reasoning mechanism because the types of messages and their semantic content shouldn't change.



Issues in developing CHI/HCHI

Stateless environments

A stateless process is one whose state never differs from its startup state; a process whose state can differ from its startup state is stateful. Any display process is thus stateful, because the display changes. Such state changes, however, do not affect the behavior of the browser mechanism. For the purposes of this paper, we distinguish state changes that can affect the behavior of the stateful mechanism as affective and those that do not as nonaffective. Changing the channel on a television set is affective; sending a different set of scan lines to its display in nonaffective. When mechanism “A” is unable to cause a behavior change in mechanism “B”, A is nonaffective with respect to B.

Page servers are nonaffective with respect to Web browsers, by design, because the behavior of the browser is totally independent of page content. No state of the server is allowed to result in anything more or less than a displayable page of text, even unrecoverable errors.

Server nonaffectiveness raises some significant issues in using a browser as a client. One is that the display of the browser is updated only at the request of the user; this is the requested-update-only problem. Another is evident in browsers that retain multiple active display states. Together, these mean that not only can the client display show incorrect information about the server state, but also that the server can get requests from the client that reflect neither its own state nor its model of the client state. In short, it's very easy for the server and the client to get de-synchronized, especially when the user enters the act. Together these result in a severe integrity problem; the browser

commands can readily cause damage to server data structures if they're inappropriate to what data are actually present.

This difficulty doesn't arise in the nominal client-server model; on the client side, the display is is slaved directly to the client, so that when the client state changes, so does the display. The client behaves just like any other standalone program. Furthermore, the client is nonaffective with respect to the server, which merely executes requests with no knowledge of client state (excepting, of course, what is embedded in the requests). Any needed state information must be retained in the client. As an aside, this is entirely satisfactory when the server is answering database queries, because all query information can be retained in the client and transmitted in the request.

This model cannot be used when the server is constructive, that is, when the server builds a data structure for the user. The server must retain state of what the user is doing; the state is, in general, too complex to include in every user/server transmission as a parameter. In standalone applications, this is not an issue because the display is slaved to the computing mechanism. In remote applications, the server can "push" information to the client when the state changes.

The [server push] facility is becoming more visible on the Web as servers gain the ability to push pages that were not specifically asked for to the browser. It is not under control of the client. Server push can be useful in, e.g., continuous update situations. You essentially get several pages for the price of one. It is also not currently possible for a server to push a page to a client unless an intial request has been made. This enables pushed pages to go right through any proxy barriers because they do have the requester tag. In particular, a server can't send a general page to a browser, then later send another page if it feels like it. This is an undesireable approach because it allows the server to gain control of the client, which is inherently dangerous. The client can abort the download, but "getting more than you asked for" is contrary to the prominent usage of the browser.

In essence, then, the problem is that a nonaffective server (by extension, all Web servers), even if it retains state and gets control of the browser, cannot affect the state of the browser. The browser, in turn, cannot retain information about the computational state to enable the usual client/server model, which assumes that state information resides in the client. Therefore, building large, complex data structures for storage, retrieval, and later use seems to be impossible.

A designer wishing to use a stateless display as an interface into a stateful process must utilize a mechanism that: (a) generates display instructions reflecting the process state (without loss of generality, meaningful interaction with a stateful process implies that its state be observable) (b) records the process state when display information goes out to the user, (c) preserves the state until a response is received, (d) when the response is received, revives the appropriate process state, (e) updates the process state as needed based on information in the response.

In a synchronous environment, b, c, and d are trivial, because the process simply waits while the user responds to the display, which is to say, the user and the computational process act as lock-step co-routines. The problem becomes more complex in the non-synchronous multi-user environment because each step must be dealt with explicitly.

CHI provides (a) - (f) transparently to the interface designer. CHI does this by creating, at display time, a unique instance that represents the page that the user is to see. CHI then sends sufficient information to uniquely identify the instance to the user's browser, along with everything the user sees. At response time (e.g., when the user submits a form that's on the page), that unique ID is returned to the CHI Web server as part of the standard HTTP [Berners-Lee95-1] packet of information (both "POST" and "GET" support this, and CHI deals correctly with both).

The receiving ACGI function dissects the HTTP message and translates the contents into CHI terms. Among the received items is the unique identifier of the page instance from which the user's browser page was generated. Since this page instance is unique, and can be accessed only by its

unique ID via this mechanism, the page instance not only preserves any relevant state information, it also provides the uniquely retrievable reconnect point for responses from this particular user to this particular state. From this page instance, all aspects of the process state that pertain to that user are available and no aspects that don't pertain to the user are available.

Addressing the issues

How an Agent uses CHI

We have designed and built a Standard Agent Framework (SAF) that enables the ready construction of classes of agents for carrying out distributed tasks. Applications to which we have applied the SAF include international electronic commerce [Goldsmith et al.], robotic simulation, and multi-agent web search. A primary component of these applications has been the standard sensory and communication interface by which agents acquire information and communicate. In this paper we focus on translation from the internal representation to HTML and back and on maintenance of state information in a connectionless, stateless communication environment..

How a developer uses HCHI

Language facilities used by CHI/HCHI
- Dispatch
- Multimethods
- Reader macros
- Live class methods
- Object Life Cycle Protocol

CHI/HCHI facilities used by the system
- sessions
- state preservation
- Runtime Class definition

In 1997 the Advanced Information Systems Laboratory (AISL) at Sandia National Laboratories completed a prototype of the Border Trade Facilitation System (BTFS), a collaborative information processing environment that operates on the Internet and World-Wide Web. The BTFS comprises multiple autonomous software agents that assist human actors in conducting international shipping transactions by creating, documenting, monitoring and coordinating shipment transactions in information space. The BTFS attacks the border-crossing problem in the three problem areas with the highest potential for improving the border-crossing process: (1) manual entry of redundant information throughout the process by different organizations; (2) incomplete regulatory documents; and (3) lack of timely status information regarding the location of the vehicle and the progress of the documentation. We discuss the conceptual design and implementation of the BTFS in the remainder of this paper.

The BTFS design is based on three general concepts: (1) creation of a distributed object programming environment with an underlying secure network infrastructure; (2) a distributed object representation of a shipping transaction; and (3) insertion of knowledgeful software agents at critical points in the information flow. Since the stakeholders in the border shipping domain are geographically distributed independent organizations, the Internet provides a ready-made communications infrastructure to integrate their operations. Using the open Internet as the communications infrastructure accommodates any and every commercial organization with access. Security is provided by public-key encryption and authentication techniques. Our initial approach suggested that the Internet, with its high ramification and ubiquity, would be well suited for the BTFS if security issues were addressed. We have solved certain security problems involving financial transactions on the Internet, but that is a topic for another forum. Otherwise, the Internet

goes well beyond merely satisfying BTFS requirements; with the BTFS in place, one could conduct international commerce from any site with an Internet connection and a web browser. The Web, nearly as far-flung as the Internet itself, also suggested HTML as the *lingua franca* of the BTFS, thus obviating the user interface dilemma and neatly solving the client end of the system. In the BTFS, a highly specialized agent converts HTML from the client into the central ontology and back.

Overlying the secure Internet is a distributed object programming system that provides a seamless design methodology for networked object environments [Spires 1997]. The distributed object system is essential to networking agents in a collaborative environment. Distributed object technology also supports a shared fragmented workpiece object. The information needed to effect a single shipment is captured in a complex distributed information structure with compositional semantics called the Maquiladora Enterprise Transaction (MET). The components of a given MET are distributed among the ECAs involved in a particular shipment. The MET is shared via proxy; when a given agent needs information in the MET, it is handed the proxy to the MET. Since the MET is distributed, no one agent or ECA has access to all components. Access is permitted based on task requirements and controlled by electronic signature. BTFS agents interact with the border-crossing process by collecting and organizing information and posting it in the MET. Control of the distributed computation is decentralized and opportunistic. Each agent computes new information components based on its internal knowledge base and the state of the MET . Changes in the components trigger computations in a manner reminiscent of blackboard systems [Englemore and Morgan 88].

[Finin *et al.* 93] Finin, T.; Weber, W.; Wiederhold, G.; Genesereth, M.; Fritzson, R.; McKay, D.; McGuire, J.; Pelavin, R.; Shapiro, S.; and Beck, C. *Specification of the KQML Agent-Communication Language -- plus example agent policies and architectures*, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993, <http://www.cs.umbc.edu/kqml/papers/kqmlspec.ps>

[Finin and Labrou 97] Finin, T. and Labrou, Y.; *A Proposal for a new KQML Specification*, TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, February 1997

[Finin *et al.* 97] Finin, T.; Labrou, Y.; and Mayfield, J.; *KQML as an agent communication language*, in Jeff Bradshaw (Ed.), “Software Agents,” MIT Press, Cambridge, 1997

[Englemore and Morgan 88] Engelmore, R., Morgan, T., (Eds.). (1988). *Blackboard Systems*. Addison-Wesley, Reading, Massachusetts.

[Goldsmith *et al.* 98] Goldsmith, S., Phillips, L., and Spires, S. (1998) A multi-agent system for coordinating international shipping, submitted to *Workshop on Agent Mediated Electronic Trading (AMET’98)*, in conjunction with Autonomous Agents ’98, Minneapolis/St. Paul, MN USA

[Spires 97] Spires, S. (1997). *The DCLOS Distributed Object System*, SNL AISL Technical Report